Problem 4 (4 points)

A tow truck is moving through a straight one-way highway from location **A** to location **B**. The driver has received many requests from customers who need to tow their cars along this highway. Each customer **C** is defined by two numbers **C.from** and **C.to**. The customer C is located along the highway at position **C.from** units away from **A**, and wants to tow his car along the highway to the location that is **C.to** units away from **A** (**C.from** < **C.to**). Customers are charged a fixed amount (100 SR) regardless of the distance. The tow truck cannot carry more than one car at the same time. Also, the driver cannot drive back.

Assume that the customer information are available in an array **C**, i.e., **C[i].from** and **C[i].to** are the pickup and the destination of the i^{th} customer, respectively.

a- Describe an algorithm that assists the tow truck driver to maximize his profit. The algorithm should print the **.from** and **.to** properties of customer cars that should be towed.

```
Tow(C[1...n])
    S[] ← ∅
    int myDist ← 0
    For i -> 1...n do
        Sort C from smallest C[].to to largest
    endfor
    For i to 1...n do
        if(C[i].from ≥ myDist)
            myDist ← C[i].to
            S ← i
            print(C[i].from, C[i].to)
        endif
    endfor
    return S
end function
```

Problem 5 (10 points)

Give the pseudo-code of an algorithm that takes as input an array **A** of integers, and returns the length of the longest contiguous subsequence of odd numbers in **A**.

Example:

The length of the longest contiguous zeros subsequence in [1, 2, 19, 5, 4, 7, 51, 23, 22, 13, 15, 36] is 3.

1 0 1 2 0 1 2 3 0 1 2 0

What is the time complexity of your algorithm? hint: Use Dynamic programming paradigm.

LCO(A[1 ... n])

S[ ] ← ∅

for i ⟶ n do

S[i] ← 0

for i=1 to n do

if (A[i] % 2 == 1)

S[i] = S[i-1] + 1

else

S[i] = 0

return max(S)

Problem 3

The Longest Decreasing Subsequence problem is defined as follows: Given a sequence of ***n*** real numbers **A[1]... A[n],** determine a subsequence (not necessarily contiguous) of maximum length in which the values in the subsequence form a strictly decreasing sequence.

Example:

The length of the longest decreasing Subsequence in [-1, **2,** -19, -5, 4, -7, -51, -2, -22, **-13,** -15, 36] is 5.

a- Give the pseudo-code of a Dynamic programming algorithm that solves the Longest Decreasing Subsequence problem.

Pseudo-code	Steps #
Proc LDS(A[1⋯n]) {	
LDS ← Ones[1⋯n]	n
for i ← 2 ⋯⋯ n do	n
for j ← 1 ⋯⋯ i−1 do	$\frac{n(n+1)}{2} - 1$
if(A[i] < A[j] and LDS[i] < LDS[j]) do	$\frac{n(n+1)}{2} - n$
LDS[i] ← LDS[j]+1	$\frac{n(n+1)}{2} - n$
return max(LDS)	1
}	

$$T(n) = \frac{3n(n+1)}{2}$$

Dijkstra's algorithm - Pseudocode

```
dist[s] ←o                                  (distance to source vertex is zero)
for  all v ∈ V–{s}
     do  dist[v] ←∞                         (set all other distances to infinity)
S←Ø                                         (S, the set of visited vertices is initially empty)
Q←V                                   (Q, the queue initially contains all vertices)
while Q ≠Ø                                  (while the queue is not empty)
do   u ← mindistance(Q,dist)                (select the element of Q with the min. distance)
    S←S∪{u}                                 (add u to list of visited vertices)
    for all v ∈ neighbors[u]
         do  if   dist[v] > dist[u] + w(u, v)            (if new shortest path found)
                then     d[v] ←d[u] + w(u, v)       (set new value of shortest path)
                     (if desired, add traceback code)
return dist
```

Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

Kruskal's Algorithm

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

SELECTION-SORT*(A)*

8	4	6	9	2	3	1

```
n ← length[A]
for j ← 1 to n - 1
    do smallest ← j
        for i ← j + 1 to n
            do if A[i] < A[smallest]
                then smallest ← i
        exchange A[j] ↔ A[smallest]
```

Algorithm 1 BinarySearch (A, *key*, *low*, *high*)

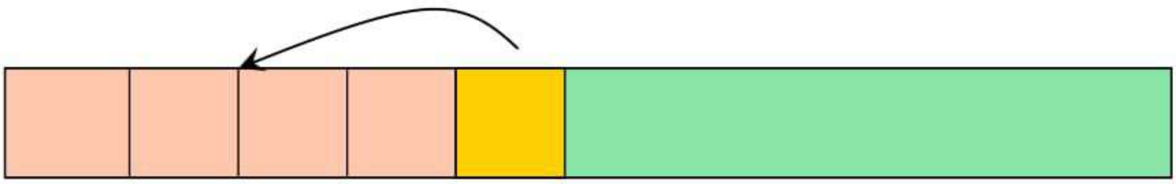
INPUT: An initially sorted array A, the target value *key*, and the starting index *low* and ending index *high* (so basically we are trying to find if A contains target value *key* within the index range $[low, high]$ of A)

OUTPUT: Index of the target value *key* within A (-1 denotes A does not contain target value *key*)

if $low > high$ **then**
    return -1.
$mid = \lfloor \frac{low+high}{2} \rfloor$;
if $value == A[mid]$ **then**
    return mid.
else if $value < A[mid]$ **then**
    return BinarySearch (A, *key*, *low*, $mid - 1$).
else
    return BinarySearch (A, *key*, $mid + 1$, *high*).

Pseudo-code

```
InsertionSort(A, n)
  for i = 2 to n do
      key = A[i]
      j = i - 1
      while  j > 0 and A[j] > key  do
             A[j+1] = A[j]
             j = j - 1
      A[j+1] = key
```



```
QuickSort(A[1 .. n]):
  if (n > 1)
      Choose a pivot element A[p]
      r ← Partition(A, p)
      QuickSort(A[1 .. r − 1])      ⟨⟨Recurse!⟩⟩
      QuickSort(A[r + 1 .. n])      ⟨⟨Recurse!⟩⟩
```

```
Partition(A[1 .. n], p):
  swap A[p] ↔ A[n]
  ℓ ← 0                 ⟨⟨#items < pivot⟩⟩
  for i ← 1 to n − 1
      if A[i] < A[n]
          ℓ ← ℓ + 1
          swap A[ℓ] ↔ A[i]
  swap A[1] ↔ A[ℓ + 1]
  return ℓ + 1
```

Figure 1.8. Quicksort

925 × 332

LCS Length Algorithm

$$c[i,j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

```
LCS-Length(X, Y)
  m = length(X)  // get the # of symbols in X
  n = length(Y) // get the # of symbols in Y
  for i = 0 to m      c[i,0] = 0       // special case: Y_0
  for j = 0 to n      c[0,j] = 0       // special case: X_0
  for i = 1 to m                       // for all X_i
      for j = 1 to n                        // for all Y_j
          if ( X_i == Y_j )
              c[i,j] = c[i-1,j-1] + 1
          else c[i,j] = max(c[i-1,j], c[i,j-1])
  return c
```

0-1 Knapsack Algorithm

- Recursive formula for subproblems:

$$B[k,w]=\begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else}\end{cases}$$

```
for w = 0 to W
    B[0,w] = 0
for i = 0 to n
    B[i,0] = 0
for i = 1 to n
    for w = 1 to W
        if w_i <= w // item i can be part of the solution
            B[i,w] = max ( b_i + B[i-1,w- w_i] , B[i-1,w] )
        else B[i,w] = B[i-1,w]  // w_i > w
```